Processo de Criação de um Periférico usando a API do QEMU

Novembro, 2023

1º: Criação de arquivos .C e .H

Qualquer periférico, sob o ponto de vista do framework do QEMU, pode ser interpretado como um *object*, seguindo o *QOM – Qemu Object Model*. Assim, para a definição e criação de um novo periférico, primeiro deve-se definir o seu tipo e, a partir deste, instanciar um objeto integrado à máquina emulada.

Para o **arquivo .C**, a primeira operação importante é a criação de um *struct* que defina o periférico e suas variáveis, inicializando seus registradores, *buffers*, ponteiros de interrupção do sistema (*irqmp*) , ponteiros que representam seu *parente object* (de quem é derivado) e variáveis de manuseio das *Memory Regions.* Na figura abaixo, pode ser vista a definição do *struct* do periférico *grlib-apbuart*:

```
struct UART {
    SysBusDevice parent_obj;

    MemoryRegion iomem;
    qemu_irq irq;

    CharBackend chr;

    /* registers */
    uint32_t status;
    uint32_t control;

    /* FIFO */
    char buffer[FIFO_LENGTH];
    int  len;
    int  current;
};
```

É possível observar a inicialização de variáveis relacionadas ao funcionamento do periférico, junto às relacionadas ao framework e APIs do QEMU em si.

Após a definição do *struct,* são definidos alguns macros que facilitam a definição do periférico no framework. O conjunto desses macros pode ser definido somente pelo *OBJECT_DECLARE_SIMPLE_TYPE(),* o qual recebe como argumento o nome do *struct* definido e o nome do tipo do objeto em *uppercase* e com termos separados por *underline*. O macro, para o código da *apbuart*, pode ser visualizado abaixo:

```
OBJECT_DECLARE_SIMPLE_TYPE(UART, GRLIB_APB_UART)
```

Após essas definições, podem ser inseridas funções de uso geral para o periférico em questão, para melhor organização do código. Dentre as funções presentes nos devices, destacam-se *device_read()* e *device_write()*, as quais são responsáveis pelo interfaceamento com a região de memória emulada do dispositivo.

Dessa forma, todos os periféricos apresentam estas funções e uma região de memória atrelada a elas. A definição das funções de *device_read()* e *device_write()* pode ser melhor observada, para o caso da *apbuart*, abaixo:

```
static void grlib_apbuart_write(void *opaque, hwaddr addr,
                                 uint64_t value, unsigned size)
{
    UART          *uart = opaque;
    unsigned char  c    = 0;

    addr &= 0xff;

    /* Unit registers */
    switch (addr) {
    case DATA_OFFSET:
    case DATA_OFFSET + 3:       /* When only one byte write */
        /* Transmit when character device available and transmitter enabled */
        if (qemu_chr_fe_backend_connected(&uart->chr) &&
            (uart->control & UART_TRANSMIT_ENABLE)) {
            c = value & 0xFF;
            /* XXX this blocks entire thread. Rewrite to use
             * qemu_chr_fe_write and background I/O callbacks */
            qemu_chr_fe_write_all(&uart->chr, &c, 1);
            /* Generate interrupt */
            if (uart->control & UART_TRANSMIT_INTERRUPT) {
                qemu_irq_pulse(uart->irq);
            }
        }
        return;

    case STATUS_OFFSET:
        /* Read Only */
        return;

    case CONTROL_OFFSET:
        uart->control = value;
        return;

    case SCALER_OFFSET:
        /* Not supported */
        return;

    default:
        break;
    }

    trace_grlib_apbuart_writel_unknown(addr, value);
}
```

```
static uint64_t grlib_apbuart_read(void *opaque, hwaddr addr,
                                   unsigned size)
{
    UART     *uart = opaque;

    addr &= 0xff;

    /* Unit registers */
    switch (addr) {
    case DATA_OFFSET:
    case DATA_OFFSET + 3:       /* when only one byte read */
        return uart_pop(uart);

    case STATUS_OFFSET:
        /* Read Only */
        return uart->status;

    case CONTROL_OFFSET:
        return uart->control;

    case SCALER_OFFSET:
        /* Not supported */
        return 0;

    default:
        trace_grlib_apbuart_readl_unknown(addr);
        return 0;
    }
}
```

Para relacionar as funções definidas com uma *MemoryRegion,* estas são definidas em um *struct MemoryRegionOps*, o qual pode ser visualizado abaixo:

```
static const MemoryRegionOps grlib_apbuart_ops = {
    .write      = grlib_apbuart_write,
    .read       = grlib_apbuart_read,
    .endianness = DEVICE_NATIVE_ENDIAN,
};
```

Além de atrelar as funções *device_read()* e *device_write()*, também é definido o *endianness* do sistema, o qual, neste caso, vem informado por um macro.

Após esta análise, vem à tona a necessidade de criação de uma classe para o objeto que representará o periférico. Normalmente, a classe gerada é derivada de uma outra mais geral. No caso da *apbuart,* a classe definida para esta é derivada de uma classe geral *DEVICE_CLASS*, a qual apresenta as funções *device_realize()* e *device_reset()* atreladas. A primeira função refere-se à chamada de algumas outras funções que fazem parte da inicialização e integração da *MemoryRegion*, interrupção e outros aspectos do device ao framework. A segunda refere-se a seu próprio *reset*, resetando seu estado e registradores. Abaixo, podem ser vistas as funções de *device_realize()* e *device_reset()* sob a ótica da *apbuart:*

```
static void grlib_apbuart_realize(DeviceState *dev, Error **errp)
{
    UART *uart = GRLIB_APB_UART(dev);
    SysBusDevice *sbd = SYS_BUS_DEVICE(dev);

    qemu_chr_fe_set_handlers(&uart->chr,
                             grlib_apbuart_can_receive,
                             grlib_apbuart_receive,
                             grlib_apbuart_event,
                             NULL, uart, NULL, true);

    sysbus_init_irq(sbd, &uart->irq);

    memory_region_init_io(&uart->iomem, OBJECT(uart), &grlib_apbuart_ops, uart,
                          "uart", UART_REG_SIZE);

    sysbus_init_mmio(sbd, &uart->iomem);
}

static void grlib_apbuart_reset(DeviceState *d)
{
    UART *uart = GRLIB_APB_UART(d);

    /* Transmitter FIFO and shift registers are always empty in QEMU */
    uart->status =  UART_TRANSMIT_FIFO_EMPTY | UART_TRANSMIT_SHIFT_EMPTY;
    /* Everything is off */
    uart->control = 0;
    /* Flush receive FIFO */
    uart->len = 0;
    uart->current = 0;
}
```

Vale ressaltar a não trivialidade da função *device_realize()*, a qual, neste caso, inicia a *MemoryRegion* referente à *apbuart*, sua interrupção em relação ao processador e relaciona a *MemoryRegion* ao *bus* do sistema.

Como essas funções são declaradas para que a classe seja derivada da *DEVICE_CLASS*, agora é necessária a chamada da função *device_class_init()*, a qual inicia a classe do próprio periférico sob a classe original. Para a *apbuart*:

```c
static void grlib_apbuart_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);

    dc->realize = grlib_apbuart_realize;
    dc->reset = grlib_apbuart_reset;
    device_class_set_props(dc, grlib_apbuart_properties);
}
```

É notável a relação entre a classe declarada e as funções *device_reset()* e *device_realize()* definidas anteriormente. Vale ressaltar que, sendo a *apbuart* um dispositivo de saída do sistema, há também a definição de algumas propriedades relacionadas a esta.

Por fim, basta a inicialização do tipo referente ao objeto e classes criados, unindo-os no struct *TypeInfo*. Para a *apbuart*:

```c
static const TypeInfo grlib_apbuart_info = {
    .name          = TYPE_GRLIB_APB_UART,
    .parent        = TYPE_SYS_BUS_DEVICE,
    .instance_size = sizeof(UART),
    .class_init    = grlib_apbuart_class_init,
};
```

É importante ressaltar que alguns dos *defines* são gerados pelo macro *OBJECT_DECLARE_SIMPLE_TYPE()*, sendo outros definidos no arquivo .h.

Para a inicialização do tipo junto ao framework, basta a definição da função *device_register_types()* e a chamada do macro *type_init()*. Para o caso da *apbuart*, tem-se:

```c
static void grlib_apbuart_register_types(void)
{
    type_register_static(&grlib_apbuart_info);
}

type_init(grlib_apbuart_register_types)
```

Essas primeiras etapas compreendem todas as implementações necessárias que devem ser feitas ao código .C do objeto. Quanto ao código .h, são definidos alguns *defines* para os nomes dos tipos dos periféricos. Para o caso dos periféricos derivados da *grlib*:

```c
/* IRQMP */
#define TYPE_GRLIB_IRQMP "grlib-irqmp"

void grlib_irqmp_ack(DeviceState *dev, int intno);

/* GPTimer */
#define TYPE_GRLIB_GPTIMER "grlib-gptimer"

/* APB UART */
#define TYPE_GRLIB_APB_UART "grlib-apbuart"

#define TYPE_GRLIB_AHBRAM "ahbram"
```

2º: Inicialização do objeto do periférico junto à máquina a ser chamada

A etapa anterior referiu-se inteiramente à definição do tipo e classe do periférico, mas não à inicialização de um objeto que o represente na máquina a ser chamada.

Para isso, os objetos dos periféricos são inicializados nas funções *machine_init()* das máquinas a serem emulados. Para o caso da *apbuart,* por exemplo, seu objeto é inicializado na função *leon3_generic_hw_init()*.

Para diferentes tipos de periféricos, podem ser necessária a chamada de diversas funções de inicialização (envolvendo a inicialização da interrupção do periférico junto ao processador do sistema, canais de saída e etc.). No entanto, algumas funções, destacam-se perante as outras estando presentes na maioria das inicializações. Para o caso da *apbuart*:

```c
/* Allocate uart */
dev = qdev_new(TYPE_GRLIB_APB_UART);
qdev_prop_set_chr(dev, "chrdev", serial_hd(0));
sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);
sysbus_mmio_map(SYS_BUS_DEVICE(dev), 0, LEON3_UART_OFFSET);
sysbus_connect_irq(SYS_BUS_DEVICE(dev), 0,
                   qdev_get_gpio_in(irqmpdev, LEON3_UART_IRQ));
grlib_apb_pnp_add_entry(apb_pnp, LEON3_UART_OFFSET, 0xFFF,
                        GRLIB_VENDOR_GAISLER, GRLIB_APBUART_DEV, 1,
                        LEON3_UART_IRQ, GRLIB_APBIO_AREA);
```

Dentre as funções presentes, as mais gerais para a maioria dos periféricos são: *qdev_new()*, que cria um novo objeto com base no tipo informado; *sysbus_realize_and_unref()*, que inicia o objeto como sendo do tipo "conectável" a um *bus* da máquina emulada; sysbus*_mmio_map()*, que mapeia a *MemoryRegion,* definida no tipo do periférico, junto à *MemoryRegion* da máquina a ser chamada.

Especificamente para a arquitetura *sparc* e para o processador *LEON3,* também há a chamada da função *grlib_apb_pnp_add_entry()*, que conecta o device ao *APB BUS* do processador do sistema.

2º: Estabelecimento de diretivas para a compilação dos arquivos adicionados

Nas etapas de *configure* e *make* dos arquivos do repositório do QEMU a fim da geração de um executável, o sistema "filtra" os arquivos e códigos a serem compilados, dependendo da arquitetura alvo escolhida.

Dessa maneira, os códigos .C e .h adicionados devem ser referenciados nas diretivas de compilação *Kconfig* e *meson.build*, arquivos presentes nas pastas onde o código .C pode ser inserido. Por exemplo, para a *apbuart:*

```
config SUN4M
    bool
    imply TCX
    imply CG3
    select CS4231
    select ECCMEMCTL
    select EMPTY_SLOT
    select UNIMP
    select ESCC
    select ESP
    select FDC_SYSBUS
    select SLAVIO
    select LANCE
    select M48T59
    select STP2000
    select CHRP_NVRAM
    select OR_IRQ

config LEON3
    bool
    select GRLIB

config GRLIB
    bool
    select PTIMER

config SLAVIO
    bool
    select PTIMER
```

A diretiva de compilação para a *apbuart* e outros dispositivos da *grlib* é *CONFIG_GRLIB*, definida no arquivo *Kconfig* presente na mesma pasta do código da inicialização do *LEON3.*

```
system_ss.add(when: 'CONFIG_CADENCE', if_true: files('cadence_uart.c'))
system_ss.add(when: 'CONFIG_CMSDK_APB_UART', if_true: files('cmsdk-apb-uart.c'))
system_ss.add(when: 'CONFIG_ESCC', if_true: files('escc.c'))
system_ss.add(when: 'CONFIG_ETRAXFS', if_true: files('etraxfs_ser.c'))
system_ss.add(when: 'CONFIG_GRLIB', if_true: files('grlib_apbuart.c'))
system_ss.add(when: 'CONFIG_IBEX', if_true: files('ibex_uart.c'))
system_ss.add(when: 'CONFIG_IMX', if_true: files('imx_serial.c'))
system_ss.add(when: 'CONFIG_IPACK', if_true: files('ipoctal232.c'))
system_ss.add(when: 'CONFIG_ISA_BUS', if_true: files('parallel-isa.c'))
system_ss.add(when: 'CONFIG_ISA_DEBUG', if_true: files('debugcon.c'))
system_ss.add(when: 'CONFIG_NRF51_SOC', if_true: files('nrf51_uart.c'))
system_ss.add(when: 'CONFIG_PARALLEL', if_true: files('parallel.c'))
system_ss.add(when: 'CONFIG_PL011', if_true: files('pl011.c'))
system_ss.add(when: 'CONFIG_SCLPCONSOLE', if_true: files('sclpconsole.c', 'sclpconsole-lm.c'))
system_ss.add(when: 'CONFIG_SERIAL', if_true: files('serial.c'))
system_ss.add(when: 'CONFIG_SERIAL_ISA', if_true: files('serial-isa.c'))
system_ss.add(when: 'CONFIG_SERIAL_PCI', if_true: files('serial-pci.c'))
system_ss.add(when: 'CONFIG_SERIAL_PCI_MULTI', if_true: files('serial-pci-multi.c'))
system_ss.add(when: 'CONFIG_SHAKTI_UART', if_true: files('shakti_uart.c'))
system_ss.add(when: 'CONFIG_VIRTIO_SERIAL', if_true: files('virtio-console.c'))
system_ss.add(when: 'CONFIG_XEN_BUS', if_true: files('xen_console.c'))
system_ss.add(when: 'CONFIG_XILINX', if_true: files('xilinx_uartlite.c'))
```

Instrução condicional de compilação para alguns arquivos representando os tipos de alguns periféricos, dentre eles a *apbuart*. Esta instrução está presente no arquivo *meson.build* localizado na mesma pasta do arquivo .C que define o tipo da *apbuart*. Vale ressaltar que esta depende diretamente da diretiva inclusa no *Kconfig* mostrada anteriormente.